AFRL-RI-RS-TR-2009-254
Final Technical Report
November 2009

PROCESS COLORING: AN INFORMATION FLOW-PRESERVING APPROACH TO MALWARE INVESTIGATION

Purdue University

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

FOR THE DIRECTOR:

/s/	/s/
JAMES L. SIDORAN	WARREN H. DEBANY, Jr.
Work Unit Manager	Technical Advisor, Information Grid Division
	Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* NOVEMBER 2009	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* June 2007 – September 2009

4. TITLE AND SUBTITLE PROCESS COLORING: AN INFORMATION FLOW-PRESERVING APPROACH TO MALWARE INVESTIGATION	**5a. CONTRACT NUMBER** N/A
	5b. GRANT NUMBER FA8750-07-2-0041
	5c. PROGRAM ELEMENT NUMBER N/A
6. AUTHOR(S) Dongyan Xu, Eugene H. Spafford and Xuxian Jiang	**5d. PROJECT NUMBER** NICE
	5e. TASK NUMBER 00
	5f. WORK UNIT NUMBER 08
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **PRIME** Purdue University, 401 South Grant Street, West Lafayette, IN 47907-2024 **SUB** North Carolina State University, 2701 Sullivan Drive, Raleigh, NC 27695	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RIGA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** N/A
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2009-254

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Process Coloring is an information-preserving, provenance-aware software system for computer malware detection and investigation. By tainting each application process with a distinct color and propagating the color to other processes or system objects along with system call operations, Process Coloring preserves the "provenance" of malware attacks (namely, "Through which process did a malware program infiltrate the system?"). Process Coloring enables three useful malware defense capabilities: (1) color-based malware detection, (2) color-based malware break-in point identification, and (3) color-based log partitioning. Implemented on top of a virtualization platform, Process Coloring achieves strong tamper-resistance as the logs generated by the protected (virtual) machine are stored and processed outside the machine under attack. Finally, Process Coloring can be integrated with techniques that track information flows inside a program. The resultant integrated system achieves better malware detection accuracy by eliminating false positive alerts, especially for client-side environments. This report gives an overview of the Process Coloring project and presents the design, implementation, and evaluation highlights in the research effort.

15. SUBJECT TERMS
Information Flow, Malware, Operating System, Virtualization

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** James L. Sidoran
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	28	**19b. TELEPHONE NUMBER (*Include area code)*** N/A

Table of Contents

1 Introduction **1**

2 Design **2**
2.1 PC for Server-side Environments . . . 3
2.2 Handling of Sinks for Client-Side Environments . . . 3
2.3 Integration with DDFA . . . 5
2.3.1 Example Scenario . . . 5
2.3.2 Color Exchange API . . . 6

3 Implementation **7**
3.1 Color Propagation . . . 7
3.1.1 When to Propagate . . . 8
3.1.2 Storing Color . . . 8
3.1.3 Shared Memory . . . 8
3.2 Logging Mechanism . . . 9
3.2.1 Xen . . . 10
3.2.2 Logging Module . . . 10
3.2.3 Log Format . . . 11
3.2.4 Log Parsing . . . 11
3.3 DDFA Integration . . . 11

4 Performance Evaluation **12**
4.1 System Performance . . . 13
4.2 Coloring Overhead . . . 14

5 Policy Evaluation **15**
5.1 Methodology . . . 15
5.2 Policy . . . 16
5.2.1 Color Order . . . 16
5.3 Scenarios . . . 16
5.4 Results . . . 18

6 Conclusion **18**

LIST OF FIGURES

1 Initial process coloring in a server machine hosting multiple services 2
2 Example of color mixing through a sink file 4
3 Unwanted color mixing occurs 6
4 Optimal color mixing occurs 7
5 Xen Architecture 9
6 Xen Split Driver Model 9
7 Process Coloring Logging Module 10
8 Color Flow Performance 15
9 Example Policy Evaluation Output 17

LIST OF TABLES

1 Log file format 12
2 Unixbench Scores with Logging 13
3 Unixbench Scores without Logging 14
4 Color Flow Custom Benchmark Results (in seconds) 14

ACRONYMS/GLOSSARY

BIND	Berkeley Internet Name Domain
COTS	Commercial Off-The-Shelf
DDFA	Dynamic Data Flow Analysis
NICECAP	National Intelligence Community Enterprise Cyber Assurance Program
DNS	Domain Name Systems
DSL	Domain-Specific Language
IP	Internet Protocol
LKM	Loadable Kernel Module
OS	Operating System
PID	Process Identifier
PC	Process Coloring
UID	User Identifier
VM	Virtual Machine
VMI	Virtual Machine Introspection
VMM	Virtual Machine Monitor

SUMMARY

Process Coloring is an information-preserving, provenance-aware software system for computer malware detection and investigation. By tainting each application process with a distinct color and propagating the color to other processes or system objects along with system call operations, Process Coloring preserves the "provenance" of malware attacks (namely, "*Through which process did a malware program infiltrate the system?*"). Process Coloring enables three useful malware defense capabilities: (1) color-based malware detection, (2) color-based malware break-in point identification, and (3) color-based log partitioning. Implemented on top of a virtualization platform, Process Coloring achieves strong tamper-resistance as the logs generated by the protected (virtual) machine are stored and processed outside the machine under attack. Finally, Process Coloring can be integrated with techniques that track information flows inside a program. The resultant integrated system achieves better malware detection accuracy by eliminating false positive alerts, especially for client-side environments. This report gives an overview of the Process Coloring project and presents the design, implementation, and evaluation highlights in the research effort.

1 INTRODUCTION

The Process Coloring project is motivated by the threats faced by cyberinfrastructures from increasingly stealthy and sophisticated malware. Recent years have witnessed numerous instances of computer malware programs that lurk in infected machines and inflict contaminations over time, such as rootkit and backdoor installation, botnet creation, and data/identity theft. Unfortunately, current log-based malware attack investigation tools (e.g., [1, 2, 3]) suffer from a number of limitations such as lack of provenance, time and space efficiency, and tamper-resistance.

Process Coloring aims at addressing the current solutions' limitations by realizing accountable information flows at the operating system (OS) level. We argue that OS-level information flows have not been fully utilized in malware defense. In particular, we have demonstrated that through the preservation of malware break-in *provenance* along OS-level information flows, we are able to improve the efficiency and effectiveness of existing intrusion investigation tools. Furthermore, a new capability of runtime malware warning not provided in existing tools can be achieved.

More specifically, Process Coloring (or "PC" for the rest of the report) is an information flow-preserving, provenance-aware approach to malware investigation. In PC, a "color," a unique system-wide identifier, is associated with every potential malware break-in point – for example, every remotely accessible service process (e.g., web, mail, or DNS server) in a server host, or every client-side application process that accepts external data (e.g., web browser, text editor, or e-mail reader). The color will be either *inherited* directly by any spawned child process, or *diffused* indirectly through processes' actions (e.g., *read* or *write* operations) along the information flows between processes or between processes and objects (e.g., files or directories). As a result, any process or object affected by a colored process will be tainted with the same color. To preserve the provenance of such influence, the corresponding log entry will also record the color. Process colors, as recorded in the log entries, reveal valuable information about possible malware break-ins and contamination actions. PC brings the following useful capabilities to malware investigation:

- *Color-based determination of malware break-in point:* All malware-affected processes and contaminated objects will be tainted with the color of the original vulnerable service/application process, the break-in point through which the malware has broken into the system. By examining the color of any malware-related log entry or any malware-affected object, the break-in point can be determined or narrowed down *before* detailed log analysis.

- *Color-based partition of log data:* The log color provides a natural index to partition the log. To reveal the contaminations caused by a malware, it is no longer necessary to examine the entire log. Instead, only those log entries carrying the color of the malware's break-in point need to be inspected. Color-based log partition substantially reduces the volume of log data to be analyzed, thereby improving the efficiency of malware intrusion re-construction.

- *Color-based malware detection:* Process coloring turns the passive log into an active generator of malware warnings based on the colors and other context information carried by the log entries. With OS-level information flow *visualized* with colors, it is possible for users/administrators to "view" the information flows at runtime and identify anomalies revealed by suspicious tainting, mixing, and relation of process (log) colors. The colors reveal anomalous influence between processes or between processes and objects under a malware attack, which is not supposed to exhibit under normal circumstances. This color-enabled capability is not provided by the existing log-based intrusion analysis tools.

- *More tamper-resistant log processing:* To achieve a more tamper-resistant implementation of malware evidence collection, process coloring leverages virtualization technology, especially virtual machine introspection (VMI) [4]. Virtualization also enables *external* (relative to the system being monitored and investigated) log processing and analysis, achieving the desirable property of non-stop system operation. We have successfully implemented PC in the Xen VMM [5].

The rest of the report is organized as follows: Section 2 presents key components in PC's design. Section 3 describes implementation details of PC. Section 4 evaluates the performance of PC, whereas Section 5 demonstrates the effectiveness of process color-based policies for malware/anomaly detection. Finally, Section 6 concludes this report.

2 Design

Process Coloring is based on a formal model that describes the diffusion of provenance information (i.e. process color) along OS-level information flows. Based on the classic information flow concept [6, 7, 8, 9], we focus on the *OS-level* information flows, where the principals are processes and the objects are system objects such as files, directories, and sockets. Our new contribution lies in the identification of provenance information (possible malware break-in points), which is defined as process color, diffused along OS-level information flows, and preserved in log entries.

Figure 1 shows an example of initial process coloring in a server machine hosting multiple services. A unique system-wide identifier called *color* is assigned to each service process. A malware trying to break into the server will have to exploit a certain vulnerability of a (colored) server process. The color of the exploited process will then be *diffused* in the host, following the actions performed by the malware. As a result, the break-in and contaminations by the malware will be evidenced by the color of the affected processes and system objects and correspondingly, by the color of the associated log entries.

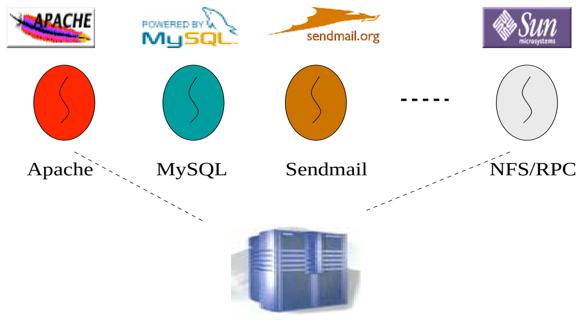


Figure 1: Initial process coloring in a server machine hosting multiple services

After the service/application processes are initially colored, the colors will be diffused to other processes along OS-level information flows among processes and system-wide shared objects. More specifically, process colors are diffused via operations performed by system calls – the OS interface that malware uses to inflict contaminations (e.g., backdoor installation).

PC requires system call interception to generate log entries and "paint" them with process colors. As demonstrated in [2, 10], system call logs are effective in revealing intrusion steps and actions. Unfortunately, the commonly used syscall hooking technique is vulnerable to the *re-hooking* attack, where an intruder easily subverts the log collection function [11]. Instead, our implementation of PC is based on the *virtual machine* *introspection* (VMI) technique [4], where the storage and processing of system call logs takes place outside the VM being protected, thus achieving stronger tamper-resistance.

Each log entry will record all "context" information of a system call (e.g., current process, syscall number, parameters, return value, return address, and time stamp), which is tagged with the color(s) of the current process. We note that the log format can be easily extended to include richer auditing information such as "who did it" (UID) and "where did it come from/go to" (IP/port).

2.1 PC for Server-side Environments

During the first six months of the project, we implemented the first version of the PC prototype and applied it to the detection and investigation of malware in server-side environments. One interesting phenomena in our server-side evaluation is *color mixing*: What does it indicate if a process (and the corresponding log entry) bears more than one color? We call such a situation "color mixing", where a different color is found diffused to an already-colored process but the process is not supposed to bear that color.

Color mixing may give away a malware attack and generate a timely alert for investigation. Based on the rationale of color diffusion, coloring mixing indicates that a process is influenced by another process with a *different provenance*. Considering the initial assignment of colors to mutually unrelated service/application processes, such cross-service/application influence is likely an anomaly (exceptions to be addressed shortly) and warrants a warning for user attention. This idea motivates us to use color mixing as a runtime malware alert generator, which reveals new properties not found in any single-color process or object.

As an example, in one of our experiments, we host BIND and Apache services in one server and let the Lion worm [12] infect the server via the BIND vulnerability. The Lion worm then contaminates the system by replacing all *index.html* files with its own defaced version. We observe that log entries recording subsequent web accesses bear the colors of *both* BIND and Apache. This is an anomaly because BIND and Apache are independent services and there should be no process that is influenced by both.

2.2 Handling of Sinks for Client-Side Environments

After obtaining good results in server-side malware defense, we proceeded to enhance PC for the detection and investigation of malware that targets vulnerable *client-side* software (e.g., web browsers).

However, when testing the PC system with client side applications, a significant amount of *unwanted* color mixing occurred. When the system was started with a few colored applications, eventually all used applications would inherit all colors. We refer to this as the *brown problem*. After extensive testing, we determined that the problem was related to *sinks* – processes or files that become both destinations and sources for large amounts of color (information) flow.

As an example, consider Figure 2, which demonstrates a sink file – `.recently_used` – in Linux's Gnome window manager. It shows a graph generated from a PC log file generated during the following scenario: The system is started with the web browser (Firefox) being colored "1" and there are no other colors in the system. The user starts Firefox and navigates to a PDF file which causes Firefox to save `file.pdf` to a temporary location, pass color "1" to it, and load the system's PDF viewer, evince. Evince inherits color "1" from the PDF file. This behavior is desirable, as it properly transmits provenance information. As evince closes, however, it writes the filename of the PDF file into `.recently_used`, a file that stores a list of documents that have been recently viewed or edited by the user. This causes "1" to be transferred to the file, even though no potentially confidential information has been leaked – We do not consider the *existence* of a file to be confidential. Later, when the system's office suite opens (soffice.bin) it reads in

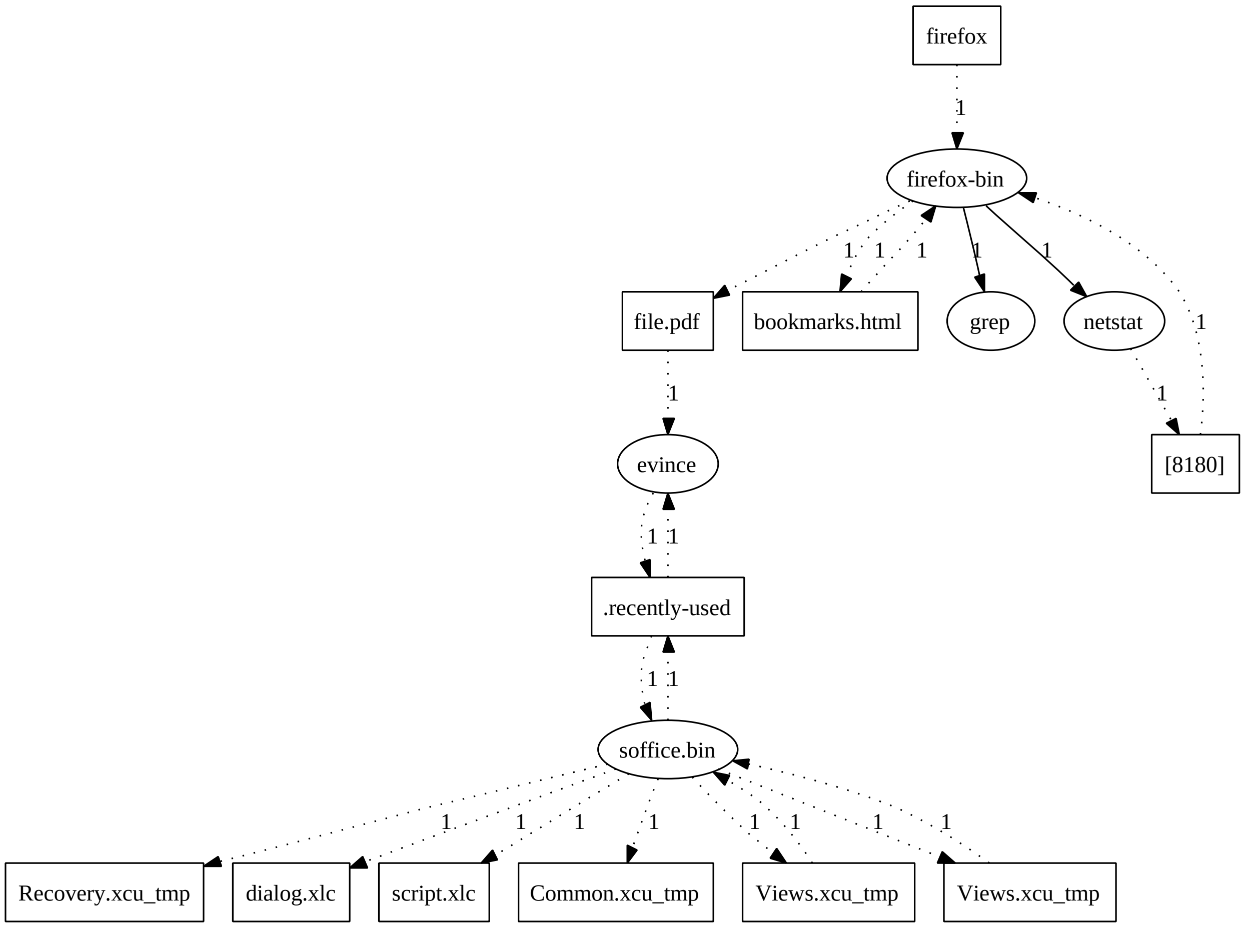


Figure 2: Example of color mixing through a sink file

`.recently_used` and inherits the color "1" as well, and all documents it saves will also inherit that color. In this case, `.recently_used` has become a sink file because it will inherit and propagate the colors of all applications which use it.

In order to address the problem of sinks we have created four potential mitigation techniques.

1. **Insulate the sink.** One approach to stopping these unwanted color flows is to ensure that the sink does not inherit or propagate color. We refer to this as insulating the sink. This technique is conceptually simple and very effective, but has some drawbacks. First, all sinks must be identified in order to ensure the brown problem is completely stopped. This requires significant profiling of all applications available on a system. Second, this opens up a vulnerability wherein an attacker can use an insulated sink to remove color from a data flow. This technique is implemented as part of the current PC implementation for testing purposes and is used to remove the brown problem so that policy testing can be conducted.

2. **Verify program context.** An extension of the insulation technique is to insulate certain color flow events. For example, we could trap all PDF viewer writes to `.recently_used` and verify that the program is writing the proper information to the file. This would ensure that an attacker couldn't hijack the program to write arbitrary information to the file and remove its color. Verification could be based on a recent chain of system calls, the application call stack, or other program level information. One problem with this technique is that even with full access to a program's memory space it is difficult to ascertain whether the program has been hijacked by an attacker. Further work in the area of remote attestation would be required to rely on this technique. We tested a preliminary version of this technique that read the application's call stack and compared it a known call stack related to writing to the sink file. If the two matched, then the color was not propagated.

3. **Strictly separate applications.** Another option is to simply restructure the operating system environment to ensure that applications do not share information through sinks. This could be as complex as redesigning all the applications and as simple as causing different applications to see different versions of the same file. One implicit goal of the PC system, however, is to work with commercial off-the-shelf (COTS) products and as such we have not pursued this method.

4. **Track program-level information flow.** The final, and we believe most rigorous technique is to track and make use of program level information flow. This is discussed in more detail in Section 2.3.

2.3 Integration with DDFA

2.3.1 Example Scenario

We have collaborated with the researchers in the Dynamic Data Flow Analysis (DDFA) project [13] – another project funded through the NICECAP program – to integrate *program level* information flow and *process level* information flow.

The DDFA system that we are integrating with is a source to source C compiler that modifies a program to track data flow within the application. It operates by propagating provenance information along internal data flows. For example, DDFA could be used to mark all information that comes from a network source as "tainted" and any new data that is created based on that data is also "tainted." Philosophically it is very similar to PC, but it operates at the application level within a program.

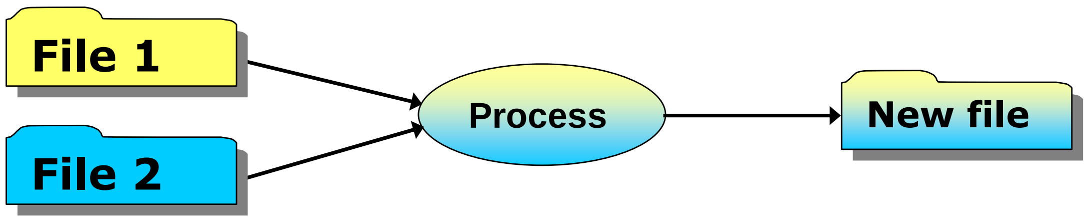


Figure 3: Unwanted color mixing occurs

In the integrated PC+DDFA system, when an application performs a read from a file, all memory that receives data from the file is marked with that file's color as received from PC. DDFA then propagates that color through the application's memory as it executes. When a later write occurs, DDFA can inform PC which of the application's colors are involved in that write and only propagate those colors.

In order to describe the DDFA+PC solution, we will start by describing a simple scenario where PC alone transmits more color than necessary. Consider Figure 3. The process is a simple text search program that reads in multiple files, searches for a keyword, and exports any lines that contain that keyword. (`grep` on Unix is an example of such a program.) In our example, let's assume that only File 2 contains the keyword. As such, the new file created by the process contains information that came only from File 2. None of File 1's data was written to the new file. PC, acting alone, would still color the file with two colors despite the fact that the data could be properly colored with only the color from File 2.

2.3.2 Color Exchange API

There are three functions that make up the API for color exchange between PC and DDFA:

1. `init_color()` – Declare to PC that this process will use the API.
2. `fetch_color(fd)` – Retrieve the color information for a given file descriptor. This function returns a 32-bit bit string.
3. `set_color(fd, color_list)` – Set the color information for a given file. Use the bit string `color_list` to determine which colors to set.

On application startup, if the application is instrumented by DDFA then it should call `init_color()` prior to reading or writing from any files. When reading from a file, the application should read as usual and then call `fetch_color(fd)` to import the color information after the read has been completed. This prevents a race condition where colored information could be written to the file and read by the application after the colors have been fetched. When writing to a file, the application should call `set_color(fd, color_list)` to export the color information prior to the write in order to ensure that no other processes can read the data before color has been associated with it.

Using this API, our sample scenario changes as seen in Figure 4. Now the process includes DDFA, which tracks provenance information within the application at a memory level. As the process performs reads on the two input files it uses `fetch_color` to taint the data read in from the files. As the application executes, that taint is tracked at runtime to ensure that any information derived from tainted data maintains the taint color. When the new file is written, DDFA knows that only the color from File 2 was used to derive the information, and uses `push_color` to inform PC of this.

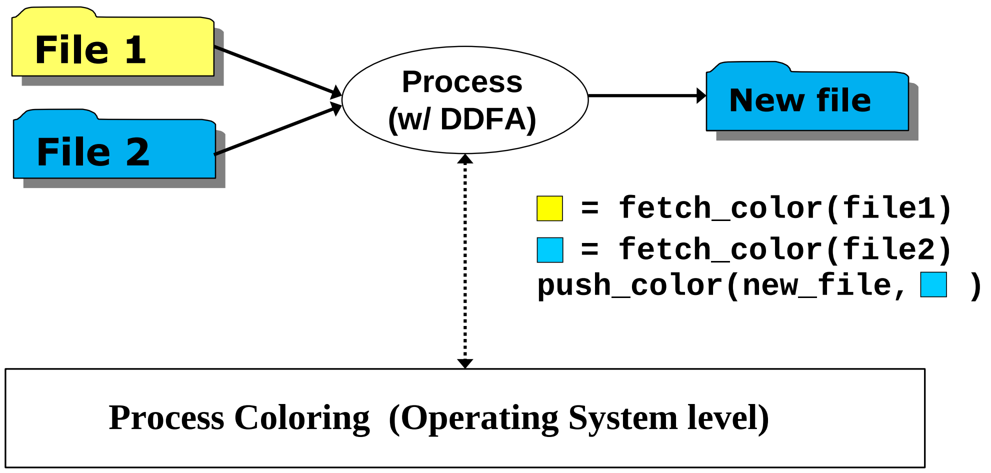


Figure 4: Optimal color mixing occurs

3 Implementation

The PC prototype was implemented in Xen 3.0.4_1 and Linux 2.6.16.33. In this section we will discuss the details of the implementation as well as the rationale behind some of the implementation decisions.

The PC implementation can be thought of in two parts. The first is a loadable kernel module (LKM) for Linux that enables color propagation for a running system. This module does not require Xen in order to operate. The second is a logging mechanism for transferring log entries related to color propagation from the guest system to the host system. The goal is to ensure that even if a guest system is compromised the log file cannot be tampered with. We do assume, however, that the Xen hypervisor itself will not be compromised.

3.1 Color Propagation

In modifying the Linux kernel to properly propagate colors during information flows, it was particularly important to ensure that all explicit data flows be captured. Our initial choice was to modify all of the relevant system calls (such as `read` and `write`) to propagate color between processes and files.

One problem with this approach, however, is that it requires the programmer to thoroughly understand all system calls and hook them appropriately. This task, while finite, is not without error. In September of 2005, for example, it was determined that the `readv`/`writev` system calls were not properly modified to add the security checks required by the Linux Security Module framework for SELinux [14]. The checks were accidentally removed while optimizing the two system calls. Despite having NSA employees and top Linux coders working on the project, the omission went unnoticed for over two years. We have no reason to believe that we would be immune from a similar error or oversight.

Given the high potential for errors in manual system call instrumentation, we chose to instead implement color propagation as a loadable kernel module that makes use of the Linux Security Module framework [15]. The LSM framework allows an LKM to supply functions to be called at various times during a data access. The framework was designed for access control, however we made use of it to propagate color during information flow. We are implicitly relying on the correctness of the framework to appropriately hook all system calls and other information flow situations. Even when one considers the `readv`/`writev` oversight above, we still believe the LSM community to be well qualified to ensure this, especially as the LSM framework has continued to mature.

3.1.1 When to Propagate

An important question to answer is where, in the code, do we need to add color propagation. The LSM framework provides a hook named `file_permission`. This function pointer is called before any reads and writes to file objects such as files and pipes. Our function, `proccolor_file_permission`, verifies that the file and process are not insulated, and then propagates color according to the access type. For a read access, color is passed from the file to the process. For a write access, color is passed from the process to the file.

It is important to note that under Linux most objects are considered "files." This includes actual files, pipes, sockets, devices, etc. This means that data flows (the exception, shared memory, is discussed below) will result in `file_permission` being called.

3.1.2 Storing Color

While the system is running, colors must be stored for both processes and file objects. The LSM framework provides `void` pointers inside various kernel objects to allow an LSM module to add data structures to them. For PC, we store a process's color entries using this pointer in its task structure. We store a file object's color entries using this pointer for the inode associated with the object.

The colors themselves are stored as a linked list ordered by color acquisition. Other types of data structures, such as bit fields, arrays, and hash tables could also be used. Increasing the efficiency of this data structure is an area of future work.

To allow colors to be persistent, we store file colors on the file system using extended attributes. Extended attributes allow the kernel to associate arbitrary information with a specific file on the filesystem. We leverage this to store color information that will persist across reboots and cache flushes. Without extended attributes, color information would only be stored in memory.

3.1.3 Shared Memory

Linux has an additional file I/O mechanism called memory mapping. When a file is memory mapped, it is connected directly into a process's page table and the process simply reads and writes from its own memory space in order to read and write the file. When two processes share memory, they both simply map the same file. In that case, the "file" may not really exist on the filesystem, but instead exist purely to facilitate shared memory.

Shared memory and memory mapped files are another method by which information may flow in a system beyond the normal file-based means. When a file (or virtual file in the case of shared memory) is mapped into a process' address space, the permissions are only checked at the time the mapping is made. This means that if a process maps file A into memory and file A is later updated to include a new color, the process could read that new information without inheriting the color. This would allow circumvention of PC.

Our current implementation does not handle this situation, however it could be modified to support it in the following way:

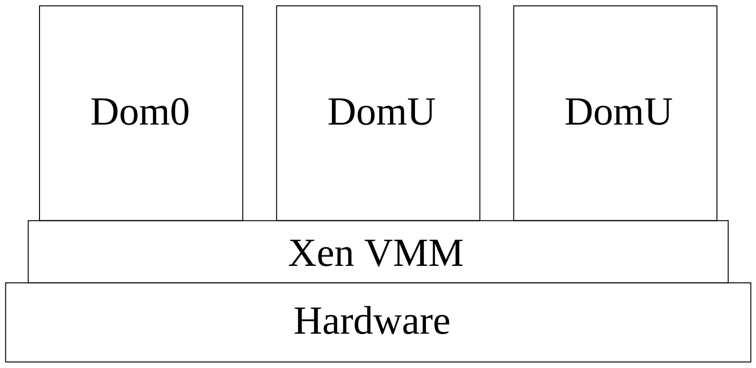


Figure 5: Xen Architecture

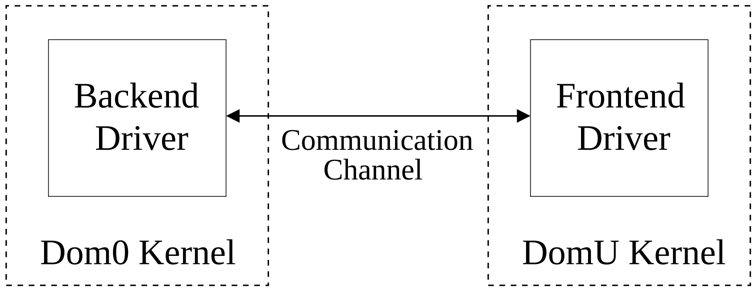


Figure 6: Xen Split Driver Model

1. When a process memory maps a file, mark the page table entry corresponding to it unavailable. This will cause a page fault on any access to that portion of the process's memory.

2. If a page fault occurs for a read access then transfer color from the file to the process. Set the page table entry to read-only. This ensures that the process cannot write to the file without our knowledge.

3. If a page fault occurs for a write then transfer color from the process to the file. Set the page table entry to readable and writable. This allows the process to continue its operation. In addition, search for any other processes which map that same file. Modify their page table entries to be unavailable. This ensures that if any other process reads from its mapping of the file, the new color can be propagated.

4. If a process inherits a new color for any reason, mark its page table entries for all mapped files to be read-only. This ensures that if a process writes to a mapped file then the new colors will be propagated.

3.2 Logging Mechanism

In this section we will discuss the mechanism used to transfer log information from the OS being monitored to the administrative host. We make use of virtualization to provide the isolation and features we need to accomplish this.

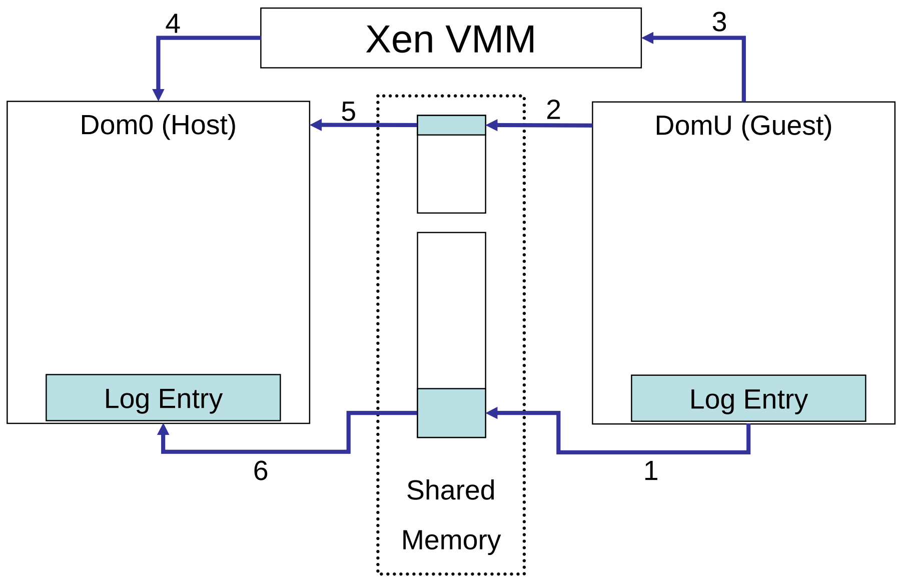


Figure 7: Process Coloring Logging Module

3.2.1 Xen

Xen [5] is a virtualization hypervisor that allows multiple operating systems to run on one physical computer. The operating systems are executed as virtual machines (VMs) while the software component that manages and schedules them is known as the virtual machine monitor (VMM). Virtual machines may also be referred to as domains. Figure 5 illustrates the Xen architecture.The Xen VMM runs directly on the physical computer and manages scheduling and resources of all running virtual machines. The first VM to startup, Dom0, has special privileges that allow it to control the other VMs (each called a DomU) on the system [16].

For privileged operations, such as those involving a network card or filesystem, Dom0 performs the operation on behalf of the DomU using a split driver model. In this model, the Dom0 contains a backend driver that interfaces with the relevant hardware while the DomU contains a frontend driver that communicates with the backend driver through a communications channel. Figure 6 illustrates this model. The Xen networking driver is a good example of this. When a packet is sent by the DomU it is sent across the communication channel to Dom0 where it is bridged to the actual ethernet card. When a packet is received for a DomU, Dom0 retrieves it from the actual ethernet card and transmits it across the communication channel to the DomU. In this way packets are shared between the less privileged DomU and the more privileged Dom0. The communication channel between Dom0 and DomU makes use of memory pages shared between the two VMs.

3.2.2 Logging Module

In order to facilitate an append-only, tamper proof log we store the log file in Dom0 and use a custom split driver to move log entries from DomU to Dom0. Log entries are generated within the DomU kernel, placed in shared memory, and then retrieved by Dom0 and made available to the user-level log processing program.

Figure 7 illustrates the custom log driver. There are six steps that correspond to the usage of the driver.

1. A log entry is generated in the DomU kernel and copied to shared memory.
2. An entry containing the location and length of the log entry is placed in an index.
3. The DomU signals the Xen VMM that data is available for Dom0.
4. The Xen VMM causes an interrupt to occur in Dom0.
5. The Dom0 interrupt service routine queries the index to look for new entries.
6. Dom0 copies the log entry from shared memory to its own memory. It then makes the entry available to the user-level log processing program by exporting it to a custom character device.

In the event that the DomU is compromised, even at the kernel level, an attacker can only modify log entries still in the shared memory or add new entries to the log. The entries already retrieved by Dom0 cannot be modified as long as the integrity of the VMM holds.

3.2.3 Log Format

The log entries themselves are designed to convey PC information flow data to the log processing program. There are 5 different types of log entries, all described in Table 1

3.2.4 Log Parsing

At the user-level in Dom0 we can run log parsers to check the log entries for compliance with a policy or simply generate usage graphs. Our log parser which generates usage graphs (Figure 2 is an example of such a graph) is written in Python and parses the log to create a graph structure which is then visualized using the Graphviz [17] software package. Our policy verification parser is written in Ruby and makes use of a small, custom policy language to specify simple policies to verify. More information about policy enforcement is available in Section 5.2.1

3.3 DDFA Integration

When integrating with DDFA it was important to create a mechanism for applications to inform and receive information about the colors involved in a specific information flow. A set of system calls, for example, could be used. One would allow DDFA to query PC after a read operation to determine which colors have flowed to the application. The other would allow DDFA to inform PC which colors were involved in a write operation after it has taken place. This would allow an application which has inherited 5 unique colors to specify that a given write is based on only 2 of them, for example.

In version 2.6 of the Linux kernel, modules are not permitted to create new system calls. As such, we opted to create a new device that a program can query using the `ioctl` system call to perform the above operations.

It is important to note that at this time the security of this solution has not been evaluated, but future research needs to investigate a method for verifying that a DDFA call to specify color flows is not maliciously attempting to cleanse color from data.

Table 1: Log file format

Fork	`type=F src=P3488 dst=P3489 p_oc="10"` A process used the `fork` system call to create a copy of itself `src` is the process id (PID) of the process which initiated the fork. `dst` is the PID of of the newly created process. `p_oc` lists the colors the process `dst` receives.
Exec	`type=E pid=3489 name="grep"` A process called the `exec` system call to execute a new binary and replace its existing memory. `pid` is the PID of the process which made the call. `name` is the name of the binary executed.
Inode Read	`type=IR src=417249 pid=3851 f_oc="15" p_oc="10,15"` A read occurred from an inode. (Inodes can represent files, pipes, or sockets.) `src` is the inode number as stored in the kernel. `pid` is the PID of the process which performed the read. `f_oc` is the list of colors the file has after the read. `p_oc` is the list of colors the process has after the read.
Inode Write	`type=IW dst=8800 pid=3851 p_oc="10,15" f_oc="10,15"` A write occurred to an inode. `dst` is the inode number as stored in the kernel. `pid` is the PID of the process which performed the write. `p_oc` is the list of colors the process has after the write. `f_oc` is the list of colors the file has after the write.
Inode Define	`type=ID src=496588 name="firefox"` This line always follows an IR or IW line. It defines the name of an inode. `src` is the inode number as stored in the kernel. `name` is the name of object associated with the inode. Pipes are named `[inode#]`.

4 Performance Evaluation

To assess the efficiency of PC, we conducted experiments using system benchmarking software. First, to measure the basic impact of Process Coloring, we used benchmark software to test system performance with and without Process Coloring installed on the system. Second, to measure the impact of coloring overhead, we used benchmark software to test system performance with Process Coloring processing an increasing number of color flows. We conducted our experiments using a single Intel® Pentium® 4 CPU 3.20GHz system running Linux kernel 2.6.16.33 and Xen 3.0.

4.1 System Performance

To establish a baseline, our system was configured with Xen and without process coloring and tested using Unixbench. After the baseline system performance was established, two additional tests were run using Unixbench in the following scenarios:

1. With Process Coloring enabled (Experiment I)
2. With Process Coloring enabled and guest-to-host logging turned off (Experiment II)

For Experiment I we assigned a color to the benchmark process itself. No other colors were assigned to the system. As a result, every benchmark related action propagated exactly one color. The biggest performance hit observed during Experiment I seemed to correspond to the creation of new processes. The evidence is shown in Table 2. Specifically, observe the overhead associated with the following tests:

Execl Throughput: Executes a file (creates new process)

Process Creation: Process creation

Shell Scripts: Shell process creation

Table 2: Unixbench Scores with Logging

Benchmark	Baseline Index Score	w/ PC Index Score	Overhead
Arithmetic Test (type = double)	245.6	243.5	0.86%
Dhrystone 2 using register variables	360.1	360.6	-0.14%
Execl Throughput	389.2	177.1	54.50%
File Copy 1024 bufsize 2000 maxblocks	772.6	732.6	5.18%
File Copy 256 bufsize 500 maxblocks	592.1	552.6	6.67%
File Copy 4096 bufsize 8000 maxblocks	760.1	1009.9	-32.86%
Pipe Throughput	414.1	378.4	8.62%
Process Creation	346.5	179.8	48.11%
Shell Scripts (8 concurrent)	659.5	106.7	83.82%
System Call Overhead	308.3	305.4	0.94%
Overall	423.5	327.3	22.72%

The PC guest-to-host logging facility is used heavily during process creation. We therefore surmised that it may be the causing the performance degradation associated with the process creation intensive tests. This led us to conduct Experiment II which confirmed our conjecture. Table 3 shows the scores associated with Experiment II. Compared with the same process creation intensive tests as previously described, the scores for each are significantly improved when logging is turned off.

We conclude from the first two experiments that the PC logging facility adds the majority of the overhead to the system according to the benchmark tests we were evaluating. Without logging, the flow of one color in a system has only 0.24% overhead.

Table 3: Unixbench Scores without Logging

Benchmark	Baseline Index Score	w/ PC no Logging Index Score	Overhead
Arithmetic Test (type = double)	245.6	247.9	-0.94%
Dhrystone 2 using register variables	360.1	370	-2.75%
Execl Throughput	389.2	398.6	-2.42%
File Copy 1024 bufsize 2000 maxblocks	772.6	712.4	7.79%
File Copy 256 bufsize 500 maxblocks	592.1	529.1	10.64%
File Copy 4096 bufsize 8000 maxblocks	760.1	1001.6	-31.77%
Pipe Throughput	414.1	360.4	12.97%
Process Creation	346.5	366.9	-5.89%
Shell Scripts (8 concurrent)	659.5	572	13.27%
System Call Overhead	308.3	315	-2.17%
Overall	423.5	422.5	0.24%

4.2 Coloring Overhead

Next we aimed to test the effects of the number of colors flowing in the system on performance. We created a program that systematically reads in 32 seed files, forks 86,528 unique processes, reads 173,056 dynamic files and writes 86,528 new unique files (pcflex.rb). The basic color flow operation involves two reads of existing colored files and one write to a new file with unique color sequence. This basic read-write operation is performed 86,528 times. The program runs for around 5 minutes when all 32 seed files are colored. To isolate the effects of the number of colors on performance, the program was run against the same number of seed files (32) each time regardless of the size of the colored subset. We performed the tests multiple times (tests 1-4 in the table) and summarized the results in Table 4. When more colors were flowing there was a larger variation in the run time, therefore we ran the benchmark process four times for 32, 16, and 8 colors and 2 times for 4, 2, 1 and 0 colors.

Table 4: Color Flow Custom Benchmark Results (in seconds)

# Colors	Avg.	Std. Dev.
32	310.24	10.5
16	297.26	3.89
8	291.48	1.76
4	291.1	6.29
2	274.15	7.29
1	263.36	4.39
0	223	4.25

As shown in Figure 8, the results show a relationship between the number of colors and the amount of time it took for the program to run. The greatest increase in run-time happens when we go from zero colors flowing to one color flowing. There appears to be a more rapid increase in run time up to four colors after which the performance impact of adding colors becomes more gradual.

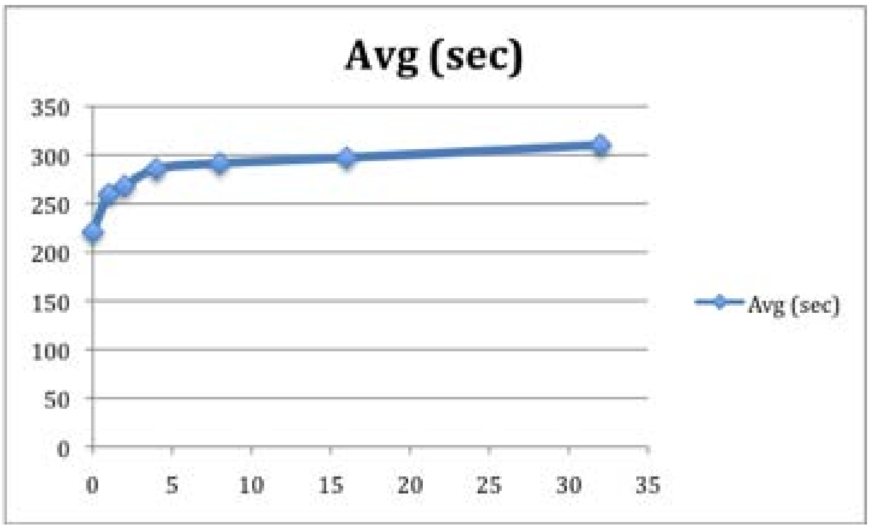


Figure 8: Color Flow Performance

5 Policy Evaluation

The Process Coloring logging facility provides a detailed trace of colors flowing in the system. Once the logging data has been obtained, it is useful to inspect it for suspicious color mixing. Initially, we used a simple policy to detect the mixing of any colors. This simple test may be sufficient for certain scenarios and color assignments. However, to clarify the sufficiency of such a policy, we conducted further experimentation as described in this section.

5.1 Methodology

As in the performance evaluation, we conducted our policy evaluation experiments using a single Intel® Pentium® 4 CPU 3.20GHz system running Linux kernel 2.6.16.33 and Xen 3.0. The guest virtual machine (DomU), where all of usage scenarios were conducted, ran on Ubuntu 6.06.1 with the same Linux kernel version as the host. DomU utilized the kernel logging mechanism to report color flow to Dom0. The logs produced by the logging mechanism were saved to Dom0's disk and evaluated offline.

Each usage scenario produced a single log file for offline evaluation. The log evaluation process consists of applying color mixing rules to each log entry. If a log entry matches a rule, a warning is displayed. The sequence of events for each scenario was as follows: Colors were assigned to each file object relating to the usage scenario. The DomU was booted. The usage scenario was executed. The DomU was terminated. The log file was then preserved for offline evaluation. Most of the scenarios were conducted during a single user session in a short amount of time (less than one hour). The "Living Lab" scenario was the exception.

To observe Process Coloring in a live environment, we used DomU as a primary desktop system for the period of one month. We refer to this setup as a "Living Lab". For this scenario, we assigned colors with the simple policy rule in mind. The simple policy rule matches any log entries with multiple colors. We were interested in discovering activities with unexpected color mixing. For example, we would be interested in a log entry that mixed the color of a network application and the color of a financial file. The tasks performed in the Living Lab were those typically undertaken by graduate researchers including: web browsing, email correspondence, and programming. To maximize the hours spent in the Living Lab environment, most activity was performed outside of the physical lab while connected remotely using a remote desktop application named Nomachine NX[18].

5.2 Policy

At the conclusion of the Living Lab usage scenario, we analyzed the resulting log file using the simple color mixing policy (matches any color mixing). Other than some unanticipated color sinks, the simple color mixing policy matched only one interesting event. It was of interest because it revealed the mixing of a web browser color and the color of a private financial file. However, upon further inspection the mixing was the result of a valid usage scenario. We refer to this scenario as the "QIF Scenario". The QIF Scenario sequence is as follows:

1. Download financial statement in the form of a Quicken QIF file from the web using browser and save it to disk.
2. Import financial statement into personal finance software [1] and save to financial records database.

The QIF Scenario exposed a weakness of our simple coloring mixing policy. The scenario is a typical and safe scenario. However, the color assigned to web browser mixed with the color assigned to the financial records database. Clearly, if the web browser transmitted the financial records database off-site that may be just cause for alarm. However, the web browser did not access the financial records database. Rather, the financial records database was influenced by a file downloaded by the web browser. This discovery lead us to investigate the possibility of color acquisition order as a policy condition.

5.2.1 Color Order

To test color order we needed a manageable way to express the order-sensitive policy conditionals. To this end we created a domain-specific language (DSL) for describing policy rules. To start, we support two clauses: WITH and FOLLOWS. The rule "10 WITH 20" would match any log entry that contains both the colors 10 and 20. The rule "10 FOLLOWS 20" would match any log entry that shows color 20 being acquired after color 10. Fortunately the Process Coloring logging facility preserves color acquisition order which makes the testing for such order possible.

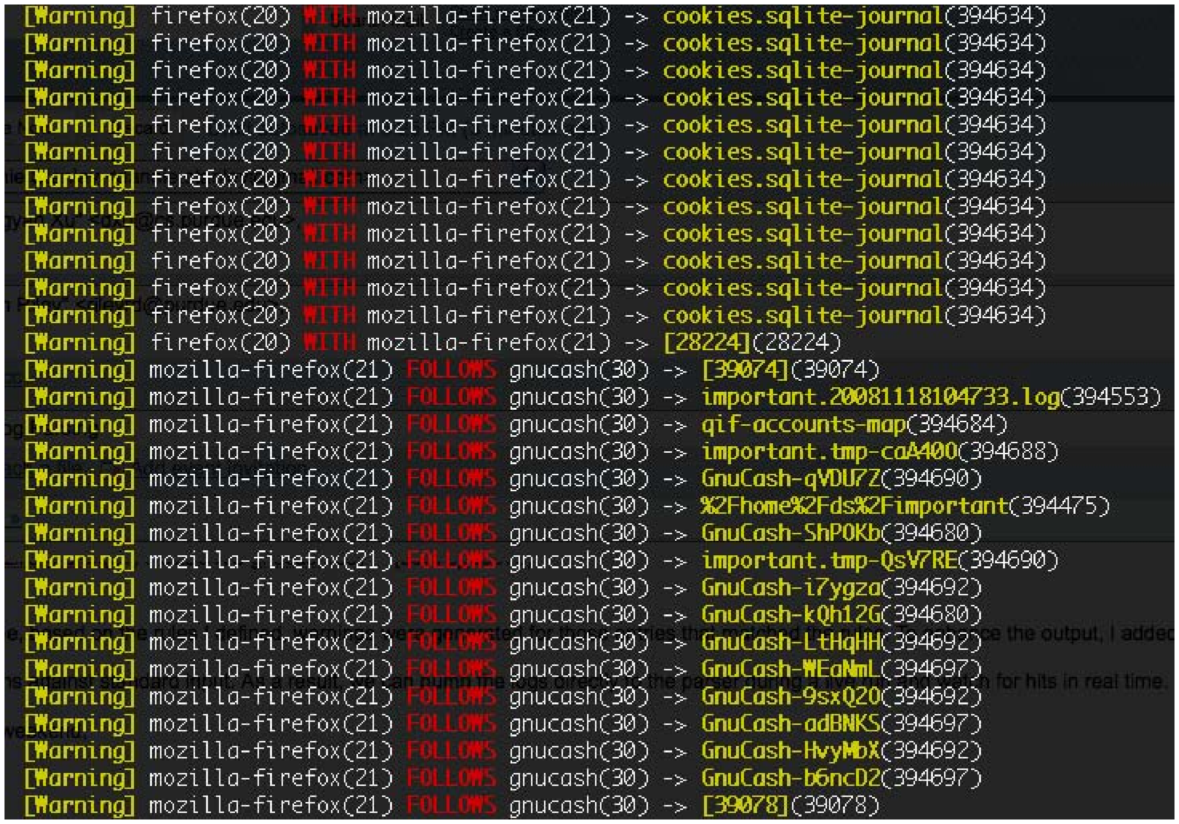
We then created a log parsing application that supports the DSL (detect.rb). The parser can be configured to match any number of rules. Additionally, it can be configured to perform a look-up that translates color ID (integer) to file name for human readable warning messages. An example of the color order policy evaluation output is shown in Figure 9.

5.3 Scenarios

We evaluated five usage scenarios using order-sensitive policies. For each evaluation we only had one policy rule defined. The rule tested the order of acquisition of the color assigned to the financial records database (35) and the color assigned to the web browser (20) [2]. The five scenarios are as follows:

[1] GnuCash: http://www.gnucash.org/

[2] In the living lab log file, the web browser is represented by two colors 20 and 21 (two versions of the same browser))



```
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> cookies.sqlite-journal(394634)
[Warning] firefox(20) WITH mozilla-firefox(21) -> [28224](28224)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> [39074](39074)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> important.20081118104733.log(394553)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> qif-accounts-map(394684)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> important.tmp-caA400(394688)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> GnuCash-qVDU7Z(394690)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> %2Fhome%2Fds%2Fimportant(394475)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> GnuCash-ShPOKb(394680)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> important.tmp-QsV7RE(394690)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> GnuCash-i7ygza(394692)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> GnuCash-kQh12G(394680)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> GnuCash-LtHqHH(394692)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> GnuCash-WEaNmL(394697)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> GnuCash-9sxQ20(394692)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> GnuCash-adBNKS(394697)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> GnuCash-HvyMbX(394692)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> GnuCash-b6ncD2(394697)
[Warning] mozilla-firefox(21) FOLLOWS gnucash(30) -> [39078](39078)
```

Figure 9: Example Policy Evaluation Output

Living Lab The Living Lab log file contains entries accumulated over a one month period during the execution of common everyday tasks. This scenario provided the motivation for investigating the order of color acquisition in policy evaluation.

QIF Scenario The QIF Scenario repeats the behavior of interest from the Living Lab scenario. The scenario was repeated to isolate the sequence of events in the log file.

Browser Scenario The Browser Scenario involves the access of the "important" financial database file by the web browser. Effectively this scenario inverts the sequence of colors found in the QIF scenario.

Agobot Scenario In the Agobot Scenario DomU is infected by the Agobot bot. We configured an isolated IRC server to communicate with the bot and executed bot commands through the corresponding IRC channel. We were able to read the financial database file using the .bot.read Agobot IRC command. The entry point for the malicious bot was the web browser. As a result, the bot carried the web browser color.

PUD Scenario In the PUD Scenario DomU is infected by the PUD bot. We configured a PUD client to access the infected DomU. We were able to read the financial database file using a remote shell command (sh 0 cat). As in the Agobot scenario, the entry point for the malicious bot was the web browser. As a result, the bot carried the web browser color.

The output of each evaluation is shown respectively below. The term "important" shown in the output refers to the financial records database file.

```
./detect.rb < ../logs/living_lab.log
./detect.rb < ../logs/qif.log
./detect.rb < ../logs/browser.log
    [Warning] important(35) FOLLOWS firefox(20) (pid = 2295)
./detect.rb < ../logs/agobot.log
    [Warning] important(35) FOLLOWS firefox(20) (pid = 3573)
./detect.rb < ../logs/pud.log
    [Warning] important(35) FOLLOWS firefox(20) (pid = 3489)
    [Warning] important(35) FOLLOWS firefox(20) (pid = 3485)
```

The absence of output from the "detect.rb" script means that no matching rule was found while processing the log file. The presence of output means that a rule was matched. Both pieces of bot software were colored with the web browser color because they were introduced to the system through the web browser. As shown in the output, no false-positives were generated.

5.4 Results

For the scenarios provided, including the Living Lab which produced much usage data, a single order-sensitive policy accurately discovered all malicious behavior and desirably filtered valid behaviors.

6 Conclusion

In this project, we have designed, implemented, and evaluated Process Coloring (PC), an information flow-preserving, provenance-aware system for malware detection and investigation. Our evaluation results indicate that PC is effective in alerting the presence of malware, identifying malware break-in points, and narrowing down system log entries for the reconstruction of malware contamination actions. The use of virtualization technology improves PC's tamper-resistance. The integration with DDFA – a program-level data flow tracking technique – effectively eliminates false positives when using PC for client-side malware and anomaly detection.

Our effort in this research is a step towards the realization of accountable information flows at multiple levels (e.g., OS level and program level) for timely, efficient, and tamper-resistant malware defense. The practicality of process coloring, which has been demonstrated in this project, can be naturally applied for other interesting security applications. For instance, the current PC prototype can be extended to support multiple physical machines and thus allow for cross-machine coloring correlation and attack attribution. The PC system is also instrumental to preventing sensitive data from being stolen, which is an essential requirement in privacy-aware applications. These opportunities remain to be explored in future research and development (R&D) endeavors.

REFERENCES

[1] Goel, A., Feng, W.-C., Maier, D., Feng, W.-C., and Walpole, J. Forensix: A Robust, High-Performance Reconstruction System. *Proc. of International Workshop on Security in Distributed Computing Systems*, June 2005.

[2] King, S. T. and Chen, P. M. Backtracking Intrusions. *Proc. of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[3] King, S. T. Mao, Z. M., Lucchetti, D. G., and Chen, P. M. Enriching Intrusion Alerts Through Multi-Host Causality. *Proc. of the 2005 Network and Distributed System Security Symposium*, February 2005.

[4] Garfinkel, T., and Rosenblum, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection . *Proc. of the 2003 Network and Distributed System Security Symposium*, February 2003.

[5] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Neugebauer, R., Ho, A., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. *Proc. of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[6] Bell, D. and LaPadula, L. MITRE Technical Report 2547 (Secure Computer System): Volume II. *Journal of Computer Security, vol. 4, no. 2/3, pages 239-263*, 1996.

[7] Clark, D. R. and Wilson, D. R. A Comparison of Commercial and Military Computer Security Policies. *Proc. of the 1987 IEEE Symposium on Security and Privacy, pages 184-194*, 1987.

[8] Denning, D. E. A Lattice Model of Secure Information Flow. *Commun. ACM 19, 5 (May), 236-243*, 1976.

[9] Goguen, J. A. and Meseguer, J. Security Policies and Security Models. *Proc. of the 1982 IEEE Symposium on Security and Privacy, pages 11-20*, 1982.

[10] Provos, N. Improving Host Security with System Call Policies. *Proc. of the 12th USENIX Security Symposium*, August 2003.

[11] Dornseif, M., Holz, T., and Klein, C. NoSEBrEaK - Attacking Honeynets. *Proc. of the 5th Annual IEEE Information Assurance Workshop, Westpoint*, June 2004.

[12] SANS Institute: Lion worm. *http://www.sans.com/y2k/lion.htm*.

[13] Chang, W., Streiff, B., and Lei, C. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 39–50. ACM New York, NY, USA, 2008.

[14] Corbet. Complete Coverage in Linux Security Modules. `http://lwn.net/Articles/154277/`, October 2005.

[15] Wright, C., Cowan, C. Morris, J., Smalley, S., and Kroah-Hartman, G. Linux Security Module Framework. In *Ottawa Linux Symposium*, 2002.

[16] Xen Project. Xen Users' Manual, Xen v3.0. `http://tx.downloads.xensource.com/downloads/docs/user/`. Last accessed July 2009.

[17] Graphviz – Graph Visualization Software. `http://www.graphviz.org/`.

[18] NoMachine NX - Desktop Virtualization and Remote Access Management Software. `http://www.nomachine.com/`.